CHAPTER 2

Exploring Containers, Classes, and Objects

In this chapter, we progress to some other essential concepts in Python - various types of containers, the methods that can be used with each of these containers, object-oriented programming, classes, and objects.

Containers

In the previous chapter, we saw that a variable could have a data type like *int*, *float*, *str*, and so on, but holds only a single value. Containers are objects that can contain multiple values. These values can have the same data type or different data types. The four built-in containers in Python are:

- Lists
- Tuples
- Dictionaries
- Sets

Containers are also called iterables; that is, you can visit or traverse through each of the values in a given container object.

In the following sections, we discuss each container and its methods in more detail.

Lists

A list contains a set of values that are stored in sequential order. A list is mutable, that is, one can modify, add, or delete values in a list, making it a flexible container.

G. Rajagopalan, *A Python Data Analyst's Toolkit*, https://doi.org/10.1007/978-1-4842-6399-0_2

An individual list item can be accessed using an index, which is an integer mentioned in square brackets, indicating the position of the item. The indexing for a list starts from 0.

Let us have a look at how to define and manipulate a list now.

Defining a list

A list can be defined by giving it a name and a set of values that are enclosed within square brackets.

CODE:

```
colors=['violet','indigo','red','blue','green','yellow']
```

Adding items to a list

Different methods can be used to add values to a list, explained in Table 2-1. The "colors" list created in the preceding code is used in the examples given in the below table.

Table 2-1. *Adding Items to a List*

Method	Description	Example

Append	Adds one item at the end of a list. The method takes only a single value as an argument.	CODE:
		`colors.append('white')`
		`#the value 'white' is added after the last item in the 'colors' list`

Insert	Adds one item at a given location or index. This method takes two arguments - the index and the value to be added.	CODE:
		`colors.insert(3,'pink')`
		`#the value 'pink' is added at the fourth position in the 'colors' list`

Extend	Adds multiple elements (as a list) at the end of a given list. This method takes another list as an argument.	CODE:
		`colors.extend(['purple','magenta'])`
		`#values 'purple' and 'magenta' added at the end of the 'colors' list`

Removing elements from a list

Just as there are multiple ways of adding an item to a list, there is more than one way to remove values from a list, as explained in Table 2-2. Note that each of these methods can remove only a single item at a time.

Table 2-2. *Removing Items from a List*

Method	Description	Example

Del	The *del* keyword deletes an item at a given location.	CODE:
		`del colors[1]`
		`#removes the second item`
		`of the 'colors' list`

Remove	This method is used when the name of the item to be removed is known, but not its position.	CODE:
		`colors.remove('white')`
		`#removes the item 'white'`
		`from the 'colors' list`

Pop	This method removes and returns the last item in the list.	CODE:
		`colors.pop()`
		`#removes the last item and`
		`displays the item removed`

Finding the index (location) of an object in the list

The *index* method is used to find out the location (or index) of a specific item or value in a list, as shown in the following statement.

CODE:

```
colors.index('violet')
```

Output:

```
0
```

Calculating the length of a list

The *len* function returns the count of the number of items in a list, as shown in the following. The name of the list is passed as an argument to this function. Note that *len* is a function, not a method. A method can be used only with an object.

CODE:

```
len(colors)
```

Output:

7

Sorting a list

The *sort* method sorts the values in the list, in ascending or descending order. By default, this method sorts the items in the ascending order. If the list contains strings, the sorting is done in alphabetical order (using the first letter of each string), as shown in the following.

CODE:

```
colors.sort()
colors
```

Output:

```
['blue', 'green', 'purple', 'red', 'violet', 'white', 'yellow']
```

Note that the list must be homogeneous (all values in the list should be of the same data type) for the sort method to work. If the list contains items of different data types, applying the sort method leads to an error.

If you want to sort your elements in the reverse alphabetical order, you need to add the *reverse* parameter and set it to "True", as shown in the following.

CODE:

```
colors.sort(reverse=True)
colors
```

Output:

```
['yellow', 'white', 'violet', 'red', 'purple', 'green', 'blue']
```

Note that if you want to just reverse the order of the items in a list, without sorting the items, you can use the *reverse* method, as shown in the following.

CODE:

```
colors=['violet','indigo','red','blue','green','yellow']
colors.reverse()
colors
```

Output:

```
['yellow', 'green', 'blue', 'red', 'indigo', 'violet']
```

Further reading:

See more on the methods that can be used with lists:

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Slicing a list

When we create a slice from a list, we create a subset of the original list by choosing specific values from the list, based on their position or by using a condition. Slicing of a list works similar to slicing a string, which we studied in the previous chapter.

To create a slice using an index, we use the colon operator (:) and specify the start and stop values for the indexes that need to be selected.

If we provide no start or stop value before and after the colon, it is assumed that the start value is the index of the first element (0), and the stop value is the index of the last element, as shown in the following statement.

CODE:

```
`colors[:]
```

Output:

```
['Violet', 'Indigo', 'Blue', 'Green', 'Yellow', 'Orange', 'Red']
```

We can also use the colon operator twice if we are using a step index. In the following statement, alternate elements of the list are extracted by specifying a step index of two.

CODE:

```
colors[::2]
```

Output:

```
['Violet', 'Blue', 'Yellow', 'Red']
```

Just like strings, lists follow both positive and negative indexing. Negative indexing (starts from –1, which is the index of the last element in the list) works from right to left, while positive indexing (starts from 0, which is the index of the first element in the list) works from left to right.

An example of slicing with negative indexing is shown in the following, where we extract alternate elements starting from the last value in the list.

CODE:

```
colors[-1:-8:-2]
```

Output:

```
['Red', 'Yellow', 'Blue', 'Violet']
```

Creating new lists from existing lists

There are three methods for creating a new list from an existing list – list comprehensions, the *map* function, and the *filter* function – which are explained in the following.

1. **List comprehensions**

   List comprehensions provide a shorthand and intuitive way of creating a new list from an existing list.

   Let us understand this with an example where we create a new list ('colors1') from the list ('colors') we created earlier. This list only contains only those items from the original list that contain the letter 'e.'

   CODE:

   ```
   colors1=[color for color in colors if 'e' in color]
   colors1
   ```

   Output:

   ```
   ['violet', 'red', 'blue', 'green', 'yellow']
   ```

The structure of a list comprehension is explained in Figure 2-1. The output expression ('color') for items in the new list comes first. Next comes a for loop to iterate through the original list (note that other loops like the *while* loop are not used for iteration in a list comprehension). Finally, you can add an optional condition using the if/else construct.

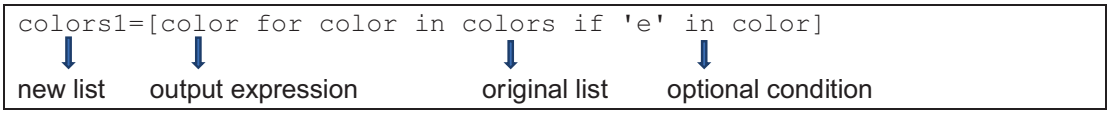

Figure 2-1. *List comprehension*

If we tried to create the same list without a list comprehension, using loops and conditions, the code would be more extended, as shown in the following.

CODE:

```
colors1=[]
for color in colors:
    if 'e' in color:
        colors1.append(color)
```

The critical point to keep in mind while using list comprehensions is that the readability of the code should not be compromised. If there are too many conditional expressions and loops involved in creating a new list, it is better to avoid list comprehensions.

Further reading: See more about list comprehensions: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

2. **Map function**

The *map* function is used to create a new list by applying a user-defined or inbuilt function on an existing list. The *map* function returns a map object, and we need to apply the list function to convert it to a list.

The map function accepts two arguments:

- The function to be applied
- The list on which the function is to be applied

In the following example, we are creating a new list ('colors1') from the 'colors' list converting its elements to uppercase. An anonymous (lambda) function is used, which is followed by the name of the original list.

CODE:

```
colors=['violet','indigo','red','blue','green','yellow']
colors1=map(lambda x:x.upper(),colors)
colors1
```

Output:

```
<map at 0x2dc87148630>
```

The function returns a map object, and the *list* function needs to be used to convert it to a list.

CODE:

```
list(colors1)
```

Output:

```
['VIOLET', 'INDIGO', 'RED', 'BLUE', 'GREEN', 'YELLOW']
```

3. **Filter function**

   The syntax of the *filter* function is similar to that of the *map* function. Whereas the *map* function returns all the objects in the original list after the function is applied, the *filter* function returns only those items that satisfy the condition specified when the filter function is called. Similar to the map function, we pass two arguments (a lambda function or a user-defined function, followed by the name of the list).

In the following example, we are creating a list from the original list, keeping only those items that have less than five characters.

CODE:

```
colors2=filter(lambda x:len(x)<5,colors)
list(colors2)
```

Output:

```
['red', 'blue']
```

Let us now discuss how we can iterate through two or more lists simultaneously.

Iterating through multiple lists using the zip function

The *zip* function provides a way of combining lists and performing operations jointly on these lists, as shown in the following. The lists that need to be combined are passed as arguments to the list function.

CODE:

```
#zip function and lists
colors=['Violet','Indigo','Blue','Green','Yellow','Orange','Red']
color_ids=[1,2,3,4,5,6,7]
for i,j in zip(colors, color_ids):
    print(i,"has a serial number",j)
```

Output:

```
Violet has a serial number 1
Indigo has a serial number 2
Blue has a serial number 3
Green has a serial number 4
Yellow has a serial number 5
Orange has a serial number 6
Red has a serial number 7
```

The *zip* function returns a list of tuples that are stored in an object of type "zip". The type of this object needs to be changed to the *list* type to view the tuples.

CODE:

```
list(zip(colors,color_ids))
```

Output:

```
[('Violet', 1),
 ('Indigo', 2),
 ('Blue', 3),
 ('Green', 4),
 ('Yellow', 5),
 ('Orange', 6),
 ('Red', 7)]
```

The next function, *enumerate*, helps us access the indexes of the items in the list.

Accessing the index of items in a list

The *enumerate* function is useful when you want to access the object as well as its index in a given list. This function returns a series of tuples, with each tuple containing the item and its index. An example of the usage of this function is shown in the following.

CODE:

```
colors=['Violet','Indigo','Blue','Green','Yellow','Orange','Red']
for index,item in enumerate(colors):
    print(item,"occurs at index",index)
```

Output:

```
Violet occurs at index 0
Indigo occurs at index 1
Blue occurs at index 2
Green occurs at index 3
Yellow occurs at index 4
Orange occurs at index 5
Red occurs at index 6
```

Concatenating of lists

The concatenation of lists, where we combine two or more lists, can be done using the '+' operator.

CODE:

```
x=[1,2,3]
y=[3,4,5]
x+y
```

Output:

```
[1, 2, 3, 3, 4, 5]
```

We can concatenate any number of lists. Note that concatenation does not modify any of the lists being joined. The result of the concatenation operation is stored in a new object.

The *extend* method can also be used for the concatenation of lists. Unlike the '+' operator, the *extend* method changes the original list.

CODE:

```
x.extend(y)
x
```

Output:

```
[1, 2, 3, 3, 4, 5]
```

Other arithmetic operators, like -, *, or /, cannot be used to combine lists.

To find the difference of elements in two lists containing numbers, we use list comprehension and the *zip* function, as shown in the following.

CODE:

```
x=[1,2,3]
y=[3,4,5]
d=[i-j for i,j in zip(x,y)]
d
```

Output:

```
[-2, -2, -2]
```

Tuples

A tuple is another container in Python, and like a list, it stores items sequentially. Like the items in a list, the values in a tuple can be accessed through their indexes. There are, however, some properties of tuples that differentiate it from lists, as explained in the following.

1. **Immutability**: A tuple is immutable, which means that you cannot add, remove, or change the elements in a tuple. A list, on the other hand, permits all these operations.
2. **Syntax**: The syntax for defining a tuple uses round brackets (or parenthesis) to enclose the individual values (in comparison with the square brackets used for a list).
3. **Speed**: In terms of speed of access and retrieval of individual elements, a tuple performs better than a list.

Let us now learn how to define a tuple and the various methods that can be used with a tuple.

Defining a tuple

A tuple can be defined with or without the parenthesis, as shown in the following code.

CODE:

```
a=(1,2,3)
#can also be defined without parenthesis
b=1,2,3
#A tuple can contain just a simple element
c=1,
#Note that we need to add the comma even though there is no element
following it because we are telling the interpreter that it is a tuple.
```

Just like a list, a tuple can contain objects of any built-in data type, like floats, strings, Boolean values, and so on.

Methods used with a tuple

While tuples cannot be changed, there are a few operations that can be performed with tuples. These operations are explained in the following.

Frequency of objects in a tuple

The *count* method is used to count the number of occurrences of a given value in a tuple:

CODE:

```
x=(True,False,True,False,True)
x.count(True)
```

Output:

```
3
```

Location/Index of a tuple item

The *index* method can be used to find the location of an item in a tuple. Using the tuple created in the previous statement, let us find the occurrence of the value, "False".

CODE:

```
x.index(False)
```

Output:

```
1
```

Only the location of the first occurrence of the item is returned by the index method.

Tuple unpacking

Tuple unpacking is the process of extracting the individual values in a tuple and storing each of these items in separate variables.

CODE:

```
a,b,c,d,e=x
```

If we do not know the number of items in a tuple, we can use the "*_" symbols to unpack the elements occurring after the first element into a list, as shown in the following.

CODE:

```
a,*_=x
print(a,_)
```

Output:

```
True [False, True, False, True]
```

Length of a tuple

The length of a tuple can be calculated using the *len* function:

CODE:

```
len(x)
```

Output:

```
5
```

Slicing of a tuple

Slicing or creation of a smaller subset of values can be performed on tuples (similar to lists and strings in this respect).

An example follows.

CODE:

```
x[::-1]
```

Output:

```
(True, False, True, False, True)
```

Applications of tuples

The following are some scenarios where tuples can be used.

1. **Creating a dictionary with tuples**

   A dictionary, which we discuss in detail in the next section, is a container containing a set of items (with a key mapping to a value). Tuples can be used for defining the items while creating a dictionary.

   A dictionary item is a tuple, and a dictionary can be defined as a list of tuples using the *dict* method, as shown in the following.

CODE:

```
x=dict([('color','pink'),('flower','rose')])
x
```

Output:

```
{'color': 'pink', 'flower': 'rose'}
```

2. **Multiple assignments**

   Tuple unpacking, as discussed earlier, is the process of separating a tuple into its components. This principle can be used for initializing multiple variables in one line, as shown in the following.

   CODE:

```
#tuple unpacking
(a,b,c,d)=range(4)
print(a,b,c,d)
```

   Output:

```
0 1 2 3
```

Further reading: See more about tuples here: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

Dictionaries

A dictionary is a container that contains a set of items, with each item mapping a "key" to a "value". Each individual item is also called a key/value pair. Some other points to note about a dictionary are:

- Unlike the values in lists and tuples, the items in a dictionary are not stored in sequential order.
- Dictionaries are mutable, like lists (i.e., one can make modifications to a dictionary object).
- Curly braces are used to enclose the items in a dictionary.

Let us understand how to define a dictionary and the different methods that can be used with a dictionary object.

Defining a dictionary

A dictionary is defined as a set of comma-separated key/value pairs enclosed within a pair of curly braces, as shown in the following code.

CODE:

```
numbers={'English':'One','Spanish':'Uno','German':'Ein'}
numbers
```

Output:

```
{'English': 'One', 'Spanish': 'Uno', 'German': 'Ein'}
```

"English", "Spanish", and "German" form the keys, while "One", "Uno", and "Ein" are the values in the dictionary.

A dictionary can also be defined using the *dict* function, as explained earlier when we discussed tuples. The argument to this function is a list of tuples, with each tuple representing a key/value pair, as shown in the following.

```
numbers=dict([('English','One'),('Spanish','Uno'),('German','Ein')])
```

Adding an item (key/value pair) to a dictionary

Using the key as an index, a new item can be added to a dictionary, as shown in the following.

CODE:

```
numbers['French']='un'
numbers
#A new key/value pair with the key as 'French' and value as 'un' has been
added.
```

Output:

```
{'English': 'One', 'Spanish': 'Uno', 'German': 'Ein', 'French': 'un'}
```

Accessing the keys in a dictionary

The *keys* method to access the keys in a dictionary:

CODE:

```
numbers.keys()
```

Output:

```
dict_keys(['English', 'Spanish', 'German', 'French'])
```

Access the values in a dictionary

The *values* method to access the values in a dictionary:

CODE:

```
numbers.values()
```

Output:

```
dict_values(['One', 'Uno', 'Ein', 'un'])
```

Access all the key/value pairs in a dictionary

The *items* method is used to access the list of key/value pairs in a dictionary.

CODE:

```
numbers.items()
```

Output:

```
dict_items([('English', 'One'), ('Spanish', 'Uno'), ('German', 'Ein'),
('French', 'un')])
```

Accessing individual values

The value corresponding to a given key can be retrieved using the key as an index, as shown in the following.

CODE:

```
numbers['German']
```

Output:

```
'Ein'
```

The *get* method can also be used for retrieving values. The key is passed as an argument to this method, as shown in the following.

CODE:

```
numbers.get('German')
```

The output is the same as that obtained in the previous statement.

Setting default values for keys

The *get* method discussed in the preceding can also be used to add a key/value pair and set the default value for a key. If the key/value pair is already defined, the default value is ignored. There is another method, *setdefault,* which can also be used for this purpose.

Note that the *get* method does not change the dictionary object, while the *setdefault* method ensures that the changes are reflected in the object.

An example of the usage of the *setdefault* method is shown in the following.

CODE:

```
numbers.setdefault('Mandarin','yi')
numbers
```

Output:

```
{'English': 'One',
 'Spanish': 'Uno',
 'German': 'Ein',
 'French': 'un',
 'Mandarin': 'yi'}
```

As we can see, a new key/value pair is added.

An example of the *get* method is shown in the following.

CODE:

```
numbers.get('Hindi','Ek')
numbers
```

Output:

```
{'English': 'One',
 'Spanish': 'Uno',
```

```
 'German': 'Ein',
 'French': 'un',
 'Mandarin': 'yi'}
```

The value set by the g*et* method is not added to the dictionary.

Clearing a dictionary

The *clear* method removes all the key/value pairs from a dictionary, in other words, it clears the contents of the dictionary without removing the variable from the memory.

```
#removing all the key/value pairs
numbers.clear()
```

Output:

```
  {}
```

Further reading: See more about dictionaries:
https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Sets

A set is a container that contains elements that are not ordered or indexed. The primary characteristic of a set is that it is a collection of *unique* elements. Note that Python does not throw an error if we add duplicate elements while creating a set. However, while we perform operations on sets, all the duplicate elements are ignored, and only distinct elements are considered.

Set definition

Just like a dictionary, a set is declared using curly braces and has unordered elements. However, while the values in a dictionary can be accessed using the keys as indexes, the values in a set cannot be accessed through indexes.

The following is an example of a set definition:

CODE:

```
a={1,1,2}
a
```

Output:

```
{1, 2}
```

As we can see from the output, the duplicate value of 1 (which is present in the set definition) is ignored.

Set operations

The methods and functions that can be used with sets are explained in Table 2-3.

Table 2-3. *Set Operations*

Operation	Method/Function	Example
Finding the length of a set	The *len* function counts the number of elements in a set, considering only the distinct values.	`len(a)`
Set iteration	The for loop can iterate through a set and print its elements.	`for x in a:` `print(x)`

Adding items or values	A single item can be added to a set using the *add* method. For adding multiple values, the *update* method is used.	`a.add(3)`
		`#or`
		`a.update([4,5])`

Removing items	Items can be removed using either the *remove* or the *discard* method.	`a.remove(4)`
		`# Or`
		`a.discard(4)`
		`#Note: When we try to delete an element that is not in the set, the discard method does not give an error, whereas the remove method gives a KeyError.`

Further reading: See more about sets: https://docs.python.org/3/tutorial/datastructures.html#sets

Now that we have covered all the essentials of the Python language - the concepts we learned in the previous chapter and what we understood in this chapter about the various containers and their methods, we need to decide which style or paradigm of programming to use. Among the various programming paradigms, which include procedural, functional, and object-oriented programming, we discuss object-oriented programming in the next section.

Object-oriented programming

Object-oriented programming (also commonly called "OOPS") emerged as an alternative to procedural programming, which was the traditional programming methodology.

Procedural programming involved sequential execution of a program using a series of steps. One major drawback of procedural programming is the presence of global variables that are vulnerable to accidental manipulation. OOPS offers several advantages vis-à-vis procedural programming, including the ability to reuse code, doing away with global variables, preventing unauthorized access to data, and providing the ability to manage code complexity.

Python follows the object-oriented paradigm. Classes and objects form the building blocks of object-oriented programming. Classes provide the blueprint or structure, while objects implement this structure. Classes are defined using the *class* keyword.

As an example, say you have a class named "Baby" with attributes as the name of the baby, its gender, and weight. The methods (or the functions defined within a class) for this class could be the actions performed by a baby like smiling, crying, and eating. An instance/object is an implementation of the class and has its own set of attributes and methods. In this example, each baby would have its unique characteristics (data) and behavior (functionality)

A class can have a set of attributes or variables, which may be either class variables or instance variables. All instances of the class share class variables, whereas instance variables are unique to each instance.

Let us see how classes are defined in Python, using the following example:

CODE:

```
#example of a class
class Rectangle:
    sides=4
    def __init__(self,l,b):
        self.length=l
        self.breadth=b
    def area(self):
        print("Area:",self.length*self.breadth)
my_rectangle=Rectangle(4,5)
my_rectangle.area()
```

Output:

```
Area: 20
```

The *class* keyword is followed by the name of a class and a colon sign. Following this, we are defining a class variable named "sides", and initializing it to 4. This variable is common to all objects of the class.

After this, there is a constructor function that sets or initializes the variables. Note the special syntax of the constructor function - a space follows the *def* keyword and then two underscore signs, the *init* keyword, again followed by two underscore signs.

The first parameter of any method defined in a class is the *self* keyword, which refers to an instance of the class. Then come the initialization parameters, "l" and "b", that refer to the length and breadth of the rectangle. These values are provided as arguments when we create the object. The instance variables, "self.length" and "self.breadth", are initialized using the parameters mentioned earlier. This is followed by another method that calculates the area of the rectangle. Remember that we need to add *self* as a parameter whenever we define any method of a class.

Once the class is defined, we can define an instance of this class, also called an object. We create an object just like we would create a variable, give it a name, and initialize it. "my_rectangle" is the name of the object/instance created, followed by an '=' sign.

We then mention the name of the class and the parameters used in the constructor function to initialize the object. We are creating a rectangle with length as 4 and breadth as 5. We then call the area method to calculate the area, which calculates and prints the area.

Further reading: See more about classes in Python: https://docs.python.org/3/tutorial/classes.html

Object-oriented programming principles

The main principles of object-oriented programming are encapsulation, polymorphism, data abstraction, and inheritance. Let us look at each of these concepts.

Encapsulation: Encapsulation refers to binding data (variables defined within a class) with the functionality (methods) that can modify it. Encapsulation also includes data hiding, since the data defined within the class is safe from manipulation by any function defined outside the class. Once we create an object of the class, its variables can be accessed and modified only by the methods (or functions) associated with the object.

Let us consider the following example:

CODE:

```
class Circle():
    def __init__(self,r):
        self.radius=r
    def area(self):
        return 3.14*self.r*self.r
c=Circle(5)
c.radius #correct way of accessing instance variable
```

Here, the class Circle has an instance variable, radius, and a method, area. The variable, radius, can only be accessed using an object of this class and not by any other means, as shown in the following statement.

CODE:

```
c.radius #correct way of accessing instance variable
radius #incorrect, leads to an error
```

Polymorphism

Polymorphism (one interface, many forms) provides the ability to use the same interface (method or function) regardless of the data type.

Let us understand the principle of polymorphism using the *len* function.

CODE:

```
#using the len function with a string
len("Hello")
```

Output:

```
5
```

CODE:

```
#using the len function with a list
len([1,2,3,4])
```

Output:

```
4
```

CODE:

```
#using the len function with a tuple
len((1,2,3))
```

Output:

```
3
```

CODE:

```
#using the len function with a dictionary
len({'a':1,'b':2})
```

Output:

```
2
```

The *len* function, which calculates the length of its argument, can take any type of argument. We passed a string, list, tuple, and dictionary as arguments to this function, and the function returned the length of each of these objects. There was no need to write a separate function for each data type.

Inheritance: Inheritance refers to the ability to create another class, called a child class, from its parent class. A child class inherits some attributes and functions from the parent class but may also have its own functionality and variables.

An example of inheritance in Python is demonstrated in the following.

```
#inheritance
class Mother():
    def __init__(self,fname,sname):
        self.firstname=fname
        self.surname=sname
    def nameprint(self):
        print("Name:",self.firstname+" "+self.surname)
class Child(Mother):
    pass
```

The parent class is called "Mother", and its attributes "firstname" and "surname" are initialized using the *init* constructor method. The child class, named "Child", is inherited from the "Mother" class. The name of the parent class is passed as an argument when we define the child class. The keyword *pass* instructs Python that nothing needs to be done for the child class (this class just inherits everything from the parent class without adding anything).

However, even if the child class does not implement any other method or add any extra attribute, the keyword *pass* is essential to prevent any error from being thrown.

Further reading: Learn more about inheritance: https://docs.python.org/3/tutorial/classes.html#inheritance

Data abstraction

Data abstraction is the process of presenting only the functionality while hiding the implementation details. For instance, a new user to Whatsapp needs to only learn its essential functions like sending messages, attaching photos, and placing calls, and not how these features were implemented by the developers who wrote the code for this app.

In the following example, where we declare an object of the "Circle" class and calculate the area using the area method, we do not need to know how the area is being calculated when we call the area method.

```
class Circle():
    def __init__(self,r):
        self.r=r
    def area(self):
        return 3.14*self.r*self.r
circle1=Circle(3)
circle1.area()
```

Output:

```
28.259999999999998
```

Summary

- A container is a collection of objects that belong to basic data types like int, float, str. There are four inbuilt containers in Python – lists, tuples, dictionaries, and sets.
- Each container has different properties, and a variety of functions that can be applied to Containers differ from each other depending on whether the elements can be ordered and changed (mutability) or not. Lists are mutable and ordered, tuples are immutable and ordered, and dictionaries and sets are mutable and unordered.
- Python follows the principles of object-oriented programming like inheritance (deriving a class from another class), data abstraction (presenting only the relevant detail), encapsulation (binding data with functionality), and polymorphism (ability to use an interface with multiple data types).
- A class contains a constructor function (which is defined using a special syntax), instance variables, and methods that operate on these variables. All methods must contain the keyword *self* as a parameter that refers to an object of the class.

In the next chapter, we will learn how Python can be applied in regular expressions and for solving problems in mathematics, and the libraries used for these applications.

Review Exercises

Question 1

How do you convert a list to a tuple and vice versa?

Question 2

Just like a list comprehension, a dictionary comprehension is a shortcut to create a dictionary from existing iterables. Use dictionary comprehension to create the following dictionary (from two lists, one containing the keys (a–f) and the other containing the values (1–6)):

```
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5}
```

Question 3

Which of the following code statements does *not* lead to an error?

a) `'abc'[0]='d'`

b) `list('abc')[0]='d'`

c) `tuple('abc')[0]='d'`

d) `dict([('a',1),('b',2)])[0]=3`

Question 4

Write a program to calculate the number of vowels in the sentence "Every cloud has a silver lining".

Question 5

What is the output of the following code?

```
x=1,2
y=1,
z=(1,2,3)
type(x)==type(y)==type(z)
```

Question 6

What is the output of the following code?

```
numbers={
    'English':{'1':'One','2':'Two'},
    'Spanish':{'1':'Uno','2':'Dos'},
    'German':{'1':'Ein','2':'Zwei'}
}
numbers['Spanish']['2']
```

Question 7

Consider the following dictionary:

```
eatables={'chocolate':2,'ice cream':3}
```

Add another item (with "biscuit" as the key and value as 4) to this dictionary using the

- If statement
- setdefault method

Question 8

Create a list that contains odd numbers from 1 to 20 and use the appropriate list method to perform the following operations:

- Add the element 21 at the end
- insert the number 23 at the 4th position
- To this list, add another list containing even numbers from 1 to 20
- Find the index of the number 15
- Remove and return the last element
- Delete the 10th element
- Filter this list to create a new list with all numbers less than or equal to 13
- Use the map function to create a new list containing squares of the numbers in the list

- Use list comprehension to create a new list from the existing one. This list should contain the original number if it is odd. Otherwise it should contain half of that number.

Answers

Question 1

Use the list method to convert a tuple to a list:

```
list((1,2,3))
```

Use the tuple method to convert a list to a tuple:

```
tuple([1,2,3])
```

Question 2

CODE:

```
#list containing keys
l=list('abcdef')
#list containing values
m=list(range(6))
#dictionary comprehension
x={i:j for i,j in zip(l,m)}
x
```

Question 3

Correct options: b and d

In options a and c, the code statements try to change the items in a string and tuple, respectively, which are immutable objects, and hence these operations are not permitted. In options b (list) and d (dictionary), item assignment is permissible.

Question 4

Solution:

```
message="Every cloud has a silver lining"
m=message.lower()
count={}
```

```
vowels=['a','e','i','o','u']
for character in m:
    if character.casefold() in vowels:
        count.setdefault(character,0)
        count[character]=count[character]+1
print(count)
```

Question 5

Output:

```
True
```

All three methods are accepted ways of defining tuples.

Question 6

Output:

```
'Dos'
```

This question uses the concept of a nested dictionary (a dictionary within a dictionary).

Question 7

Solution:

```
eatables={'chocolate':2,'ice cream':3}
#If statement
if 'biscuit' not in eatables:
    eatables['biscuit']=4
#setdefault method(alternative method)
eatables.setdefault('biscuit',4)
```

Question 8

Solution:

```
odd_numbers=list(range(1,20,2))
#Add the element 21 at the end
odd_numbers.append(21)
#insert the number 23 at the 4th position
```

```
odd_numbers.insert(3,23)
#To this list, add another list containing even numbers from 1 to 20
even_numbers=list(range(2,20,2))
odd_numbers.extend(even_numbers)
#find the index of the number 15
odd_numbers.index(15)
#remove and return the last element
odd_numbers.pop()
#delete the 10the element
del odd_numbers[9]
#filter this list with all numbers less than or equal to 13
nos_less_13=filter(lambda x:x<=13,odd_numbers)
list(nos_less_13)
#use the map function to create a list containing squares
squared_list=map(lambda x:x**2,odd_numbers)
#use list comprehension for the new list
new_list=[x/2 if x%2==0 else x for x in odd_numbers]
new_list
```